PIE: A Dynamic Failure-Based Technique*

Jeffrey M. Voas, *Member, IEEE*

Abstract

This paper presents a dynamic technique for statistically estimating three program characteristics that affect a program's computational behavior: (1) the probability that a particular section of a program is executed, (2) the probability that the particular section affects the data state, and (3) the probability that a data state produced by that section has an effect on program output. These three characteristics can be used to predict whether faults are likely to be uncovered by software testing.

Index Terms: Software testing, data state, fault, failure, testability.

1 Introduction

Software testing has several advantages over other verification forms: it relies on less formal analysis than a technique such as proof of correctness, it replicates operational behavior, and it has a statistical basis. However, software testing has drawbacks: any predictions based on software testing depend on an assumed input distribution. If the assumed input distribution is inaccurate, or if the input distribution changes over time, any predictions based on software testing can be invalidated. When testing reveals a failure, it provides little help in locating the fault. Finally, testing requires an oracle; since automated oracles are rarely available, human oracles who require time and sometimes get the wrong results (which misleads testers) are required.

The technique described in this paper complements software testing. The technique, called *propagation, infection, and execution* analysis (or *PIE* analysis) [25], is closely related to fault-based testing [16, 15, 17, 18, 3, 24, 27] and mutation testing [4, 5, 2, 22, 7, 19, 20]. *PIE* analysis is distinct from both mutation testing and fault-based testing because *PIE* analysis collects information concerning the semantics of the program; fault-based testing collects information concerning whether certain classes of faults exist in a program; and mutation testing judges whether sets of inputs are adequate at catching faults. The technique *PIE* analysis does *not* reveal the existence of faults; correctness is never an issue. Nor does the technique directly evaluate the ability of inputs to reveal the existence of faults. Instead, *PIE* analysis identifies locations in a program where faults, *if they exist*, are more likely to remain undetected during testing. A location in *PIE* analysis is: an assignment, input statement, output statement, or the <condition> part of an **if** or **while** statement.

*This paper was written while author was supported by a National Research Council Nasa-Langley Resident Research Associateship. Since writing this article, author as accepted a position at: Reliable Software Technologies Corporation, Penthouse Suite, 1001 N. Highland Street, Arlignton, VA 22201.

PIE analysis requires no oracle and no specification. No oracle is needed because this is not a verification technique. *PIE* analysis does require an input distribution. *PIE* analysis uses program instrumentation, syntax mutation, and changed values injected into data states to predict a location's ability to cause program failure if the location were to contain a fault. The program inputs are selected at random consistent with an assumed input distribution. This analysis does not require a testing oracle because *PIE* analysis uses the program itself as an oracle for examining the output of altered versions of the program.

The technique measures the effect that a location of the program has on the program's dynamic computational behavior when:

1. The program is executed with inputs selected from a particular input distribution. This estimates the frequency with which inputs execute the location.

2. The location is mutated via syntactic mutants. This estimates the frequency with which mutants of the location create altered data states.

3. The data state created by the location has a value in it changed. This estimates the frequency with which altered data states cause a change in the program's output.

Since these scenarios dynamically simulate the three necessary and sufficient conditions for software failure to occur, *PIE* analysis is a dynamic failure-based technique.

The remainder of this paper is organized as follows: §2 describes the theoretical model that *PIE* analysis is based on; the *PIE* analysis technique that implements this model is presented in §3. §4 presents an application of *PIE* analysis, and §5 presents results using the *PIE* analysis technique.

2 Theoretical Model

2.1 General Definitions

We view a program as an implementation of a function g that maps a domain of possible inputs to a range of possible outputs. Another function f with the same domain and perhaps different range represents the desired behavior of g. An *oracle* is a recursive predicate on input-output pairs that checks whether or not f has been implemented for an input, i.e., oracle $\omega(x, y)$ is TRUE iff $f(x) = y$. Then the oracle is used with $g(x)$ for y. During testing, it is necessary to be able to say whether a particular output of a program is *correct* or *incorrect* for a particular input x, with the latter implying that $g(x) \neq f(x)$, and the former implying that $g(x) = f(x)$. The *failure probability* of program P, with respect to an input distribution D, τ_{PD}, is the probability that P produces an incorrect output for an input selected at random according to D.

In *PIE* analysis, it is necessary to uniquely identify specific syntactic program constructs as well as the internal data states created during execution. To uniquely identify syntactic constructs, we define a *location* to be either an assignment statement, an input statement, an output statement, or the <condition> part of a **if** or **while** statement. Our definition for location is based on Korel's [10] definition for a single instruction.

A program *data state* is a set of mappings between all variables (declared and dynamically allocated) and their values at a point during execution; in a data state we include both the program input used for this execution and the value of the program counter. We only identify

data states between two dynamically consecutive locations. The execution of a location is considered here to be atomic, hence data states can only be viewed between locations. As an example, the data state

$$\{(\text{input}, 3), (\text{a}, 5), (\text{b}, 5), (\text{c}, \text{undefined}), (\text{pc}, 10)\}$$

tells that variables **a** and **b** have the value 5, the next instruction to be executed is at address 10, variable **c** is undefined, and the program input that started this execution was a 3. Before program execution on an input begins, all variables are undefined.

A *data state error* is an incorrect variable/value pairing in a data state where correctness is determined by an assertion between locations. We refer to a data state error as an *infection*, and use these two terms interchangeably. If a data state error exists, the data state and variable with the incorrect value at that point are termed *infected*. A data state may have more than one infected variable. *Propagation* of a data state error occurs when a data state error affects the output. *Cancellation* has occurred when the existence of a data state error is not discernible in the program output, i.e., after viewing the output, we have no indication that a data state error ever occurred. Cancellation is commonly referred to as *coincidental correctness*. In this paper, we do not look at intermediate locations for data state error cancellation; only at output locations.

If there exists at least one input from a distribution D for which a program P fails, then we say that P contains a *fault* with respect to D. Even though we may know that a fault exists in a program, we cannot in general identify a single location as the exclusive cause of the failure. For example, several locations may interact to cause the failure, or the program can be missing a required computation which could be inserted in many different places to correct the problem. However, if a program is annotated with assertions about the correct data state before and after a particular location l, and if there exists an input from D such that l's succeeding data state violates the assertion and l's preceding data state does not violate the assertion, then l contains a fault.

In *PIE* analysis, it is important to be able to determine whether a particular variable at some specific location of a program has any potential impact on the output computation of the program. A variable is termed *live* at a particular location if this potential exists. Determination of whether a variable is live is made statically from a flow graph that is augmented with def-use information. Admittedly, certain variables defined as live via static analysis using a flow graph containing def-use information might not be defined as live if our definition were based on the dynamic behavior of the program [9].

2.2 Model Definitions

This section formalizes:

1. The probability that a location is executed—an execution probability.

2. The probability that a change to the source program causes a change in the resulting internal computational state—an infection probability.

3. The probability that a forced change in an internal computational state propagates and causes a change in the program's output—a propagation probability.

To define execution, infection, and propagation probabilities, we first introduce notation. Recall this technique tracks data states as a program executes. It is therefore necessary to uniquely identify a data state according to the input that the program is currently executing on, the last location executed in the program where we are observing the data state, and which iteration of the location we are observing this data state on (if the location is executed more than once for this input).

Let S denote a specification, P denote an implementation of S, x denote a program input, Δ denote the set of all possible inputs to P, D denote the probability distribution of Δ, l denote a program location in P, and let i denote a particular execution (or what we term an "iteration") of location l caused by input x. Let $\mathcal{B}_{lPix}$ represent the data state that exists prior to executing location l on the i^{th} execution from input $x \in \Delta$, and let $\mathcal{A}_{lPix}$ represent the data state produced after executing location l on the i^{th} execution from input x.

It is important for us to be able to group data states into sets with similar properties. For instance, assume that location l is executed n_{xl} times by input x. Then we might want to look at all of the data states that are created by this input immediately before l is executed or immediately after l is executed. The following sets allow us to do so:

$$\mathcal{B}_{lPx} = \{\mathcal{B}_{lPix} \mid 1 \leq i \leq n_{xl}\}$$

$$\mathcal{A}_{lPx} = \{\mathcal{A}_{lPix} \mid 1 \leq i \leq n_{xl}\}$$

We further group these sets for all $x \in \Delta$:

$$\beta_{lP\Delta} = \{\mathcal{B}_{lPx} \mid x \in \Delta\}$$

$$\alpha_{lP\Delta} = \{\mathcal{A}_{lPx} \mid x \in \Delta\}$$

We let f_l denote the function that *is* computed at a location l. The input to a function computed at a location is a data state and the output of such a function is also a data state. Thus

$$\mathcal{B}_{lPix} \xrightarrow{f_l} \mathcal{A}_{lPix}.$$

The *execution probability* ε_{lPD} of location l of program P is simply the probability that a randomly selected input x selected according to D will execute location l.

Let $\mathcal{M}_l$ represent a set of z_l mutants of location l: $\{m_{l1}, m_{l2}, ..., m_{lz_l}\}$ (where $1 \leq y \leq z_l$) [22, 7, 19, 20]. And let $f_{m_{ly}}$ denote the function computed by mutant m_{ly}. The *infection probability* $\lambda_{m_{ly}lPD}$ of mutant m_{ly} is the probability that the succeeding data state of location l is different than the succeeding data state that mutant m_{ly} creates, given that l and m_{ly} execute on a data state that would normally precede l (one that would be created by a randomly selected input according to D).

We define a *simulated infection* as a changed value forced into the value of some variable (that already had a value) in a data state. As we have already stated, $\mathcal{A}_{lPix}$ denotes the data state created after the i^{th} iteration of location l on input x; $\breve{\mathcal{A}}_{lPix}$ denotes this same data state after a simulated infection is injected into $\mathcal{A}_{lPix}$. A simulated infection affects a single live variable.

The *propagation probability* ψ_{ailPD} for a simulated infection affecting variable a in the i^{th} data state succeeding location l (where this data state is created by a randomly selected input x according to D) is the probability that P's output differs (from what would normally be produced) after execution is resumed using the simulated infection.

3 Estimations of the Theoretical Model

We can estimate the previous three probabilities using a set of program inputs that are selected at random according to D. Three analyses are used for estimating the execution probability, infection probability, and propagation probability: Execution analysis, Infection analysis, and Propagation analysis. These methods are the focus of §3 and collectively are termed *PIE* analysis.

Before we describe these analyses, we assume several properties about any program undergoing *PIE* analysis as well as knowledge about the program's environment:

1. The program is close to being correct, meaning that it compiles and is believed to be close to a correct version of the specification both semantically and syntactically; this essentially is the *competent programmer hypothesis* [22]. If this property is not met, *PIE* analysis will still deliver estimates; however, the estimates will be of less significance. The closeness is required because the confidence in the applicability of the resulting estimates diminishes as we move further away from the assumption.

2. A distribution of inputs, D, is available from which we can sample.

3. The inputs that we sample are only from Δ.

4. The cardinality of Δ is effectively infinite for sampling purposes. Although there are finitely many numbers representable on a computer, we will assume this fixed number exceeds what can be exhaustively sampled from during testing.

3.1 Execution Analysis

Execution analysis is a method that is based on program structure. As such, execution analysis is related to structural testing methods. Structural testing methods attempt to cover specific types of software structure with at least one input. For example, *statement testing* is a structural testing method that attempts to execute every statement at least once; *branch testing* is a structural testing method that attempts to execute each branch at least once. *Execution analysis* estimates the probability of executing a particular location when inputs are selected according to a particular input distribution.

Statement testing and branch testing are weak criteria because their satisfaction does not ensure failure should a fault exist. Executing a statement during statement testing and not observing program failure merely provides *one* data point for estimating whether or not the statement contains a fault. Execution analysis benefits structural testing methods by indicating the likelihood of executing a particular statement.

Execution Analysis estimates execution probabilities. The *execution estimate* of execution probability ε_{lPD} is denoted by $\hat{\varepsilon}_{lPD}$—it is found by finding the proportion of inputs (selected according to D) that cause location l is executed. As will be discussed further in §3.5, a potential exists for inputs that are selected according to D to appear to cause non-terminating computations. For this reason, if a non-terminating computation is suspected, a mechanism within execution analysis will be required that will terminate execution analysis on that input (meaning that input will be ignored). This situation may cause the resulting execution estimates to be a function of some input distribution other than D, but regardless, such a mechanism is needed.

3.2 Infection Analysis

Infection analysis is similar to the processes employed in fault-based testing. *Fault-based testing* aims at demonstrating that certain faults are not in a program. Morell [18, 16, 15, 17, 3] proves properties about fault-based strategies concerning certain faults that can and cannot be eliminated using fault-based testing. Since fault-based testing restricts the class of possible faults, the possible testing is limited. Fault-based testing defines faults in terms of their syntax.

Fault-based testing also evaluates inputs based on their ability to distinguish the specific faults. *Mutation testing* [22, 7, 19, 20] is a fault-based testing strategy that does just this—it evaluates program inputs. It takes a program P and produces n versions (*mutants*) of P, $[p_1, p_2, ..., p_n]$, that are syntactically different from P. The goal of *strong mutation testing* [22] is to find a set of inputs that distinguishes the mutants from P.

Another variant of mutation testing, *weak mutation testing* [7], selects inputs that cause all imagined infections to be created by a possibly infinite set of mutants. Infection analysis statistically measures the effect a set of mutants have on data states. Syntactic changes are made to program locations and infection analysis finds the probability that a particular mutant affects a data state.

Infection analysis estimates an infection probability for each mutant $m_{ly} \in \mathcal{M}_l$. The estimate of some infection probability $\lambda_{m_{ly}lPD}$ is termed an *infection estimate* and is denoted by $\hat{\lambda}_{m_{ly}lPD}$—it is found by the following algorithm:

1. Set variable **count** to 0.
2. Randomly select an input x according to D, and if P halts on x in a fixed period of time, find the corresponding $\mathcal{B}_{lPx}$ in $\beta_{lP\Delta}$. (§3.5 explains how to handle the possibly non-recursive and infinite nature of $\beta_{lP\Delta}$). Uniformly select a data state $\mathcal{Z}$ from $\mathcal{B}_{lPx}$.
3. Present the original location l and the mutant m_{ly} with data state $\mathcal{Z}$ and execute both locations in parallel.
4. Compare the resulting data states and increment **count** when $f_l(\mathcal{Z}) \neq f_{m_{ly}}(\mathcal{Z})$.
5. Repeat steps 2-4 n times.
6. Divide **count** by n yielding the sample mean of $\frac{\text{number of times that } f_l(\mathcal{Z}) \neq f_{m_{ly}}(\mathcal{Z})}{n}$; this is our $\hat{\lambda}_{m_{ly}lPD}$.

The mutants that have been used in this research have been limited to mutants of arithmetic expressions and predicates. For arithmetic expressions, the mutants considered in our research are limited to single changes to a location—this is similar to the mutations used in mutation testing [22, 7, 19, 20]. Our assignment statement mutants are: (1) a wrong variable substitution, (2) a variable substituted for a constant, (3) a constant substituted for a variable, (4) expression omission, and (5) a wrong operator. For boolean predicates, our mutants are: (1) substituting a wrong variable, (2) exchanging **and** and **or**, and (3) substituting a wrong equality/inequality operator. We have purposely limited the syntactic changes to single changes to avoid the explosion that occurs in the number of combinatorial changes that could be made at each location.

One difficulty with mutants is determining semantic equivalence between the mutant and the original location. In infection analysis, we have handled this problem as follows: If we ever receive a 0.0 infection estimate, we statically trace the code to attempt to determine whether any data state $\mathcal{Z}$ will ever exist in the $\mathcal{B}_{lPx}$s such that $f_{m_{ly}}(\mathcal{Z}) \neq f_l(\mathcal{Z})$. If we do determine semantic equivalence, we discard the mutant from the set and ignore its infection estimate. If we determine that they are not semantically equivalent, we allow the infection estimate to stand. If we are unable to make a determination due to circumstances such as code complexity, we allow the infection estimate to stand.

In summary, infection analysis reveals statistical information about the effect that mutants have on data states. All of the problems associated with generating mutants in mutation testing exist in infection analysis as well. In this initial stage of our research, we have closely followed the mutation techniques developed by [22, 7, 19, 20]; as our experience with *PIE* analysis increases, we expect to gain insight into the strengths and weaknesses of different mutation techniques.

3.3 Propagation Analysis

In this section, we discuss using simulated infections to predict the propagation of actual infections (if they exist). *Propagation analysis* estimates propagation probabilities. The *propagation estimate* of propagation probability ψ_{ailPD} is denoted by $\hat{\psi}_{ailPD}$—it is found by the following algorithm:

1. Set variable **count** to 0.

2. Randomly select an input x according to D, and if P halts on x in a fixed period of time, find the corresponding $\mathcal{A}_{lPx}$ in $\alpha_{lP\Delta}$. Note that in practice, we expect that P halts on each input x in a fixed (and reasonable) amount of time. If for any input x, P does not halt in this allocated time, we ignore x and will not use any data state $\mathcal{A}_{lPix}$ in this algorithm. (§3.5 explains how to handle the possibly non-recursive and infinite nature of $\alpha_{lP\Delta}$). Set $\mathcal{Z}$ to $\mathcal{A}_{lPix}$.

3. Alter the sampled value of variable a found in $\mathcal{Z}$ creating $\breve{\mathcal{Z}}$, and execute the succeeding code on both $\breve{\mathcal{Z}}$ and $\mathcal{Z}$. Possible methods for altering the value of variable a are discussed below.

4. For each different result in program output after termination on $\breve{\mathcal{Z}}$ and $\mathcal{Z}$, increment **count**; increment **count** if a time limit for termination related to the altered state has been exceeded. The time limitation should be a function of the time for completion required using the non-altered state $\mathcal{Z}$. This precaution is necessary because of the effects that altered variables can cause to boolean conditions that terminate indefinite loops. Of course we cannot be certain that termination using the altered state $\breve{\mathcal{Z}}$ will not eventually occur, however we must set some time limit or this algorithm might never terminate.

5. Repeat steps 2-4 n times.

6. Divide **count** by n yielding the sample mean of $\frac{\text{number of times that program output differed}}{n}$; this is our $\hat{\psi}_{ailPD}$.

In propagation analysis, a simulated infection is created by a perturbation function. The process of injecting a simulated infection is termed *perturbing*. A *perturbation function* is a mathematical function that takes in a data state as an incoming parameter, changes it according to certain parameters that are either input to the function or hard-wired, and produces as output a different data state. A data state that has had a value changed by a perturbation function is said to have been *perturbed*. We only perturb live variables within a data state because we already know from the definition of live that perturbing a variable in a data state which is not live will result in a 0.0 propagation probability.

Perturbation functions can create a wide variety of simulated infections by using a pseudo-random number generator—we use the Lehmer pseudo-random number generator with a fixed initial seed described in [13]. To perturb, we actually insert the necessary code to cause a state perturbation into the program under analysis. We do so by inserting a source-code module containing the pseudo-random number generator into the code under analysis, and place a call to this module from the location where we want a data state perturbed. We send the module the current data state value and the module returns the perturbed data state value. To date, we have only perturbed numeric data state values; perturbing non-numeric data state values is an area of future research.

The decision concerning when during execution to inject a simulated infection is important to the resulting propagation estimates. That is, during which iteration or iterations of a location do we apply a perturbation function? For example, if a location is in a loop that iterates three times, then we can inject a simulated infection on any of the following combinations of iterations: (1), (2), (3), (1, 2), (1, 3), (2, 3), (1, 2, 3). Note that if we do decide to inject a simulated infection on more than one execution of a location, the simulated infection affects the same variable on each iteration. We currently do not perturb on combinations of variables, due to the explosion in the number of potential combinations. This too is an area for future research.

The choice of how and when to apply a simulated infection depends on the type of data state error we are simulating. Recall that propagation analysis simulates the occurrence of data state errors, and it is important that the simulated infection mimic the real-world, i.e., we must simulate the types of data state errors that actual faults create. For example, since faults in a conditional location can affect which branch is taken after the conditional location, we include the program counter as a live variable; this allows us to perturb the program counter by using an enumerated type (whose members are the different locations that could be executed after the conditional location is executed) and randomly selecting a member of the type as the perturbed program counter value. As another example, if the type of data state error being simulated can be mapped to a type of fault that has a tendency to produce a data state error each time a fault from this class is executed, then a perturbation function is applied in $\mathcal{A}_{lPix}$ for each i. An example is off-by-one faults, which always infect when executed. In general, mapping simulated infections to potential actual faults will not be possible, since potential faults are very difficult to determine. So in our research experiments we have perturbed on each iteration (meaning we apply a perturbation function to the current data state value even if that value was a result of a previous perturbation) since our experience has shown that faults frequently infect on each iteration.

To handle perturbing on each iteration of a location, we define a variant of the previously defined propagation probability; this variant handles the case where a simulated infection is injected into a variable on every iteration of a location. ψ_{alPD} is the probability that P's output

differs given that the value of variable a is perturbed in each data state succeeding location l.

Possible distributions for perturbation functions include all continuous and discrete distributions. To date, we have used a uniform distribution because of our lack of knowledge as to which distribution is best if "a best" exists. Also, the uniform distribution has given encouraging results that are presented in §5.

3.4 Understanding the Resulting Estimates

When *PIE* analysis is completed for the entire program, we have three sets of probability estimates for each program location l in P given a particular distribution D:

1. Set 1: Execution estimate—the estimate of the probability that program location l is executed.

2. Set 2: Infection estimates—the estimates of the probabilities, one estimate for each mutant in $\mathcal{M}_l$ at program location l, that given the program location is executed, the mutant will adversely affect the data state.

3. Set 3: Propagation estimates—the estimates of the probabilities, one estimate for each live variable at program location l, that given that the variable in the data state following location l changes, the program output that results changes.

Note that each probability estimate has an associated confidence interval, given a particular level of confidence and the value of n used in the algorithms. The computational resources available when *PIE* analysis is performed will determine an n for each algorithm. For example, for 95% confidence, the confidence interval is approximately $p \pm 2\sqrt{p(1-p)/n}$, where p is the sample mean [21, 8]. Since the ns used in the algorithms are expected to be large, $2\sqrt{p(1-p)/n}$ will likely be insignificant.

It should be noted that *PIE* analysis is a technique that can suffer from qualitative errors and thus confidence intervals play a minor role in any confidence in the probability estimates. This is because we are making rough approximations via perturbation functions and mutants. Our confidence in the value of these approximations is not a result of 95% or 99% confidence intervals, but rather because the approximations have been shown experimentally to often reflect the effect actual faults cause.

3.5 Feasibility of Implementing the PIE Algorithms

The feasibility of *PIE* analysis as a practical method lies in the ability of the infection analysis and propagation analysis algorithms to sample internal data states. The sets:

1. $\mathcal{B}_{lPix}$

2. $\mathcal{A}_{lPix}$

3. $\mathcal{B}_{lPx}$

4. $\mathcal{A}_{lPx}$

5. $\beta_{lP\Delta}$

6. $\alpha_{lP\Delta}$

are not necessarily recursive, since no algorithm exists that will tell us whether P will halt on an arbitrary input x [26]. The sets, however, may be partially computable, since we can sometimes find a subset of Δ for which P halts on each element. (It is unsolvable to decide whether or not we can find such a subset.) Since these theoretical problems can impose serious practical limitations on *PIE* analysis, this section describes a mechanism by which infection analysis and propagation analysis can sometimes get the data states they need even though the above sets are not necessarily recursive.

It is not possible to determine whether an arbitrary input causes a particular location to be executed. If possible, the famed halting problem would be solved. But we can select a specific input, execute the program on the input, and if the program halts in some fixed period of time that we have preset, we can determine whether the input executed a particular location. Thus assuming we are able restrict Δ to a finite subset denoted by δ by already knowing that each member of δ is an input on which P halts in some fixed period of time that we set, the sets

1. $\mathcal{B}_{lPix}$, where $x \in \delta$
2. $\mathcal{A}_{lPix}$, where $x \in \delta$
3. $\mathcal{B}_{lPx}$, where $x \in \delta$
4. $\mathcal{A}_{lPx}$, where $x \in \delta$
5. $\pi_{lP\delta}$, where $\pi_{lP\delta} = \{\mathcal{B}_{lPx} \mid x \in \delta\}$
6. $\varpi_{lP\delta}$, where $\varpi_{lP\delta} = \{\mathcal{A}_{lPx} \mid x \in \delta\}$

are recursive. Thus in practice, the algorithms in §3 will need to sample data states from these six sets. So Step 2 in the infection analysis algorithm is replaced by:

2. Uniformly select an input x in δ, and find the corresponding $\mathcal{B}_{lPx}$ in $\pi_{lP\delta}$. Uniformly select a data state $\mathcal{Z}$ from this $\mathcal{B}_{lPx}$.

and Step 2 in the propagation analysis algorithm is replaced by:

2. Uniformly select an input x in δ, and find the corresponding $\mathcal{A}_{lPx}$ in $\varpi_{lP\delta}$. Set $\mathcal{Z}$ to $\mathcal{A}_{lPix}$.

In practice then, *PIE* analysis becomes a function of a set of inputs, selected at random according to D, on which P halts in a fixed period of time. Unfortunately, this means that propagation estimates and infection estimates may be a function of an input distribution that is far different than D. As already mentioned, this situation may also impact execution estimates if we find that during execution analysis we must restrict the inputs used because some of the inputs selected according to D appear to be causing non-terminating computations.

Our scheme for generating δ is simple: Set a program execution time limit and sample k inputs according to D from Δ. To determine if a particular input x from the k inputs should be in δ, execute P on x and keep account of the time that passes during execution. If the amount of time used equals the time limit and termination has not occurred, do not include x in δ. This does not mean that P will not halt on x, but rather that we are not going to wait to find out.

If P halts within the time limit for some x, x is added to δ. We apply this method for each of the k inputs.

As execution analysis is performed, data states are created from which infection analysis and propagation analysis could sample if the data states were stored. Assuming that execution analysis selects inputs on which our program halts in the fixed amount of time mentioned earlier, then execution analysis generates members of $\varpi_{lP\delta}$ and $\pi_{lP\delta}$; it just does not store them.

If we were able to store the data states that occur during execution analysis, it is not necessarily the case that we will generate as many data states during execution analysis as the infection analysis and propagation analysis algorithms need for the desired level of confidence in the estimates (See §3.4). Those locations that were infrequently executed during execution analysis will have few data states stored for them.

For infrequently executed locations, we have 3 options for dealing with the small number of data states available for infection analysis and propagation analysis:

1. Continue to execute the program during execution analysis until enough data states are created.

2. Ignore the location during propagation analysis and infection analysis.

3. Follow a heuristic like Korel's [11] for finding test data that executes the location.

The first option is not practical; the second option is practical but limits the information produced by the technique, and the third option has the potential to bias the resulting estimates if the heuristic generates inputs that do execute the location but are not in Δ. Therefore, we consider option 2 as the only practical option.

Because data states are needed by both infection analysis and propagation analysis, we recommend that execution analysis be performed before the other algorithms. Even if we cannot store the data states created during execution analysis, information concerning the costs of getting a data state during infection analysis and propagation analysis is immediately available from the execution estimates. In practice, we believe that propagation analysis and infection analysis is only viable for frequently executed locations.

4 Sensitivity Analysis

The remainder of the paper focuses on making predictions. Our goal is to show how the information of *PIE* analysis can complement software testing.

We say that a fault can more easily *hide* from software testing when the fault's effect on the computation is difficult to discern. §4 shows (1) how to apply the propagation, infection, and execution estimates in order to make predictions about where faults can more easily hide and (2) how to quantify the number of tests necessary to be convinced that a location is not hiding a fault from detection. Note the shift in emphasis here—from estimation (that *PIE* analysis performs) to prediction.

Sensitivity analysis (*SA*) predicts where faults can hide [12]. Sensitivity analysis uses *PIE* analysis's estimates to predict the *minimum* effect on the failure probability that a particular location would have if a fault were present, i.e., *SA* ranks program locations based on their ability to impact the program's computation.

With sensitivity analysis, a framework is created for addressing the following questions: (1) Where can we get the maximum benefit from limited testing resources? (2) When should we use another validation technique other than testing? (3) What degree of testing must be performed to persuade ourselves that a location is probably not hiding a fault? (4) When should we rewrite the software in a manner that makes it less likely to hide faults?

Sensitivity of a location l is a prediction of the minimum probability that a fault in l will result in a software failure under a particular program input distribution. If location l is assigned a sensitivity of 1.0 under a particular input distribution D, then it is predicted that each input in D that executes l will result in a software failure if l were to contain a fault. If l is assigned a sensitivity of 0.0 under D, then it is predicted that no matter what fault is present in l, no input in D that executes l will cause a failure. (Note that there is a continuum of sensitivities in [0,1].) The greater the likelihood that a fault in location l will be revealed during testing, the greater the sensitivity that is assigned to l. A location with a low sensitivity is termed *insensitive*. A location with a high sensitivity is termed *sensitive*.

Testing either reveals or does not reveal faults; *SA* quantifies the significance of testing when testing reveals no faults. If testing's goal is to estimate the probability of failure, sensitivity *is not* an issue. Sensitivity *is* only an issue when testing's goal is to reveal faults. *SA* allows us to gauge how much trust we can place in testing for faults.

As an example, consider a simple program P:

```
{SPECIFICATION: output 1 if a^2 + b^2 + c^2 < 900000 else output 0}
(1) read(a);
(2) read(b);
(3) read(c);
(4) d := sqr(a) + 1000;
(5) e := sqr(b);
(6) f := sqr(c);
(7) if ((d + e + f) < 900000) then
(8)       writeln("1")
    else
(9)       writeln("0");
```

P is supposed to perform the function described in the braces but contains a fault in location 4. The fault in location 4 is the addition of 1000 to variable **d**. Assume that testing P under a particular input probability distribution D produces no failures. What does this testing say about the existence of faults in P? While we can make predictions about P's probability of failure under D, we do not have any assurances about an absence of faults in P. This is because: (1) the tests selected from D may not execute portions of P where the faults (if any) reside, (2) incorrect data states may not be produced, or (3) incorrect data states may be cancelled.

For P, Figure 1(i) and Figure 1(ii) contain input probability distributions that are not likely to reveal the fault in location 4. If the range of potential input values for variables **a**, **b**, and **c** were fixed in the interval [545, 550], then the fault is more likely to be caught during testing. *SA* would warn that locations 4, 5, and 6 have a greater capacity to hide faults when testing is performed according to the input probability distributions in Figure 1(i) or Figure 1(ii).

Figure 1: Probability distributions that are unlikely to reveal a fault at location 4, 5, or 6.

PIE analysis's probability estimates for P given that Figure 1(i) or Figure 1(ii) were used follow:

1. Execution analysis reveals that there is a zero probability of a fault existing in locations 4, 5, and 6 and not being executed, and thus produce $\hat{\varepsilon}_{4PD} = 1.0$, $\hat{\varepsilon}_{5PD} = 1.0$, and $\hat{\varepsilon}_{6PD} = 1.0$.

2. Infection analysis reveals that locations 4, 5, and 6 produce high infection estimates, suggesting that actual faults in the locations would almost certainly produce infections.

3. Unlike the high probability estimates of infection analysis and execution analysis, propagation analysis tells something quite different. Propagation analysis will produce low propagation estimates for variables **d**, **e**, and **f** at locations 4, 5, and 6.

Mapping the probability estimates for a location to a single sensitivity for that location is difficult, because determining the relative importance for each different set of probability estimates is difficult. Since each estimate has an associated confidence interval, we take the lower bound on the confidence interval. This assures that if bias occurs when finding a sensitivity, the bias causes underestimation of the sensitivity rather than overestimation. This conservative approach is taken one step further by only considering the minimum of the lower bounds of the infection estimates and the minimum of the lower bounds of the propagation estimates of a location when determining the location's sensitivity.

We take the conservative approach because we would rather label a location as "unsafe" (likely to hide faults) when it is actually "safe" (not likely to hide faults) than to label an unsafe location as safe. Believing that a location is more capable of revealing a fault when untrue is the negative effect associated with overestimated probability estimates that we mentioned earlier. This is the reason that a narrower interval is advocated for the perturbation function parameters—to lessen the likelihood of overestimated propagation estimates.

Let $(\cdot)_{min}$ denote the lower bound for the confidence interval for an estimate, and let $(\cdot)_{max}$ denote the upper bound for the confidence interval for an estimate. The scheme for mapping *PIE* analysis's probability estimates into a sensitivity prediction of some location l, denoted by θ_l, assuming that location l was executed frequently enough to obtain propagation and infection estimates follows:

$$\theta_l = (\hat{\varepsilon}_{lPD})_{min} \cdot \sigma(\min_{m_{ly}}[(\hat{\lambda}_{m_{ly}lPD})_{min}], \min_{a}[(\hat{\psi}_{ailPD})_{min}]) \tag{1}$$

where

$$\sigma(a,b) = \begin{cases} a-(1-b) & \text{if } a-(1-b) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Equation 1 represents a prediction of the minimum probability that if a fault were to exist in l, the existence of the fault will be revealed through testing. Note that in equation 1, $\min_a[(\hat{\psi}_{alPD})_{min}]$ may be substituted for $\min_a[(\hat{\psi}_{ailPD})_{min}]$. In this equation, we subtract the minimum proportion of data states that did not propagate when perturbed, i.e., $1-\min_a[(\hat{\psi}_{ailPD})_{min}]$ from the proportion of data states, $\min_{m_{ly}}[(\hat{\lambda}_{m_{ly}lPD})_{min}]$, that did cause an infection by mutant m_{ly}. We then take the result and multiply multiply it by the lower bound on the execution estimate for the location. This is a very conservative approach to sensitivity—this approach is taken to avoid the possibility that the data states represented in the proportion $\min_{m_{ly}}[(\hat{\lambda}_{m_{ly}lPD})_{min}]$ are the data states represented in the proportion $(1-\min_a[(\hat{\psi}_{ailPD})_{min}])$ [14].

For locations that were not frequently executed, and thus did not receive infection or propagation analysis, we cannot directly apply equation 1. We therefore can only warn that these are locations of high insensitivity, and present the execution estimate as proof of this insensitivity.

With sensitivities in hand, we now return to the four proposed questions and explain how *SA* can begin to answer them.

1. *Where to get the most benefit from limited testing resources:*
   Sensitive locations require less testing than insensitive locations. By identifying sensitive locations, sensitivity analysis saves resources that can be applied to more critical insensitive locations.

2. *When to use some other validation technique other than testing:*
   Sensitivity analysis may show extreme insensitivity (greater potential for the hiding of faults), thereby pinpointing locations for which no reasonable amount of testing under an assumed distribution can be performed to gain confidence in such locations' lack of faults. At such locations, alternative techniques should be applied such as testing under a new distribution, proofs of correctness, code review, symbolic testing [16, 15, 17, 18, 3], or exhaustive testing.

3. *The degree to which testing must be performed in order to be convinced that a location is probably not protecting a fault from detection:*
   Sensitivity analysis results may be used to determined how many test cases are necessary to be convinced a location is correct with an acceptable confidence. θ_l can be used as an estimate of the minimum failure probability for location l in the equation $1-(1-\theta_l)^T = c$ [6], where c is the confidence that the actual failure probability of location l is less than θ_l. With this equation, we can obtain the number of tests T needed for a particular c. To obtain confidence c that the true failure probability of a location l is less than θ_l given the sensitivity of the location, we need to conduct T tests that execute the location, where

$$T = \frac{\ln(1-c)}{\ln(1-\theta_l).} \tag{2}$$

4. *Whether or not software should be rewritten:*
   Sensitivity analysis results may be used as a guide to whether critical software has been sufficiently tested. If a piece of critical software is classified as having many insensitive locations, then the software may be rejected since too much testing will be required to achieve a sufficient level of confidence from testing.

5 An Experiment Comparing *PIE* Estimates to Failure Probabilities

We present evidence here that propagation analysis and execution analysis accurately estimate the effect that faults have on the failure probability of a program.

The experiment proceeds in two stages:

1. We inject a fault into a correct program. We assume a certain input distribution and establish an estimate of the failure probability of the (now faulty) program via random

software testing. For the results shown here, we purposely injected faults with probabilities of creating data state errors of approximately 1.0 so that the likelihood of low probabilities of creating data state errors affecting the failure probability was negligible. The faults used were removing an assignment statement or changing an operator in an arithmetic expression.

2. We perform propagation analysis and execution analysis on the faulty program, and use the propagation analysis and execution analysis estimates to predict a probability of failure for the location where we injected the fault.

Our hypothesis is that there will be a significant correlation coefficient between the estimate of the probability of failure measured by random software testing and the probability of failure predicted by the estimates of propagation analysis and execution analysis.

The results reported here are based on the gold-version, G, of a battle simulation that was approximately 2000 lines in length and is specified in [23]. The experiment proceeded as follows:

1. Make a copy of G, denoted by G'.
2. Randomly select some location l of G.
3. Inject a fault F into G' at location l.
4. Find $\hat{\tau}_{G'D}$, the failure probability estimate for G', using random software testing with distribution D.
5. Find $\hat{\varepsilon}_{lG'D}$, the execution estimate for location l in G'.
6. Find $\hat{\psi}_{alG'D}$, the propagation estimate for location l in G', where a is the variable on the left-hand side of the assignment statement at l in G. The $\hat{\psi}_{alG'D}$s are a function of:
   (a) A perturbation function producing a uniformly selected value in the interval $[0.5x, 1.5x]$, where x is the value variable a had before being perturbed. The Lehmer pseudo-random number generator used is described in [13]. §3.3 tells how we handle the situation where $x = 0$.
   (b) A uniform program input distribution.
   (c) 100 program inputs.
7. Go back to step 1.

Using faults with probabilities of creating data state errors of approximately 1.0 allowed us to calculate the correlation coefficient between $(\hat{\varepsilon}_{lG'D} \cdot \hat{\psi}_{alG'D})$ and $\hat{\tau}_{G'D}$ (see Table 1 and Table 2 for more specific details of the experiment). We multiply the execution estimate and the propagation estimate because propagation estimates are conditioned on executing a location.

The results presented here used a perturbation function containing a uniform distribution with (0.5·current_value) and (1.5·current_value) as parameters; these parameters were chosen *ad hoc* to get quick initial feedback on propagation analysis. We are careful when using this distribution and parameters to make sure that the result is different than current_value, and if current_value is originally 0, that we randomly generate a value that is close to 0 but not 0.

Although this perturbation function gave encouraging results, the topic of creating perturbation functions requires more research.

The reason for the success of the uniform(0.5·current_value, 1.5·current_value) perturbation function is not completely clear. On possible reason is that the potential values that result from this distribution are in a narrow interval. We conjecture that the greater the distance between a correct and incorrect data state value, the greater the probability of propagation; thus a narrower interval lessens the probability of the propagation estimates being biased upwards. This conjecture is yet unsubstantiated.

Using the correlation coefficient formula $\frac{\mathrm{Cov}(Q,Y)}{\sqrt{\mathrm{Var}(Q)\mathrm{Var}(Y)}}$, where $\mathrm{Cov}(Q,Y)$ is the covariance of random variables Q and Y, and $\mathrm{Var}(Q)$ is the variance of random variable Q [1], the correlation coefficient between the probability of failure predicted by propagation analysis and execution analysis and the estimate of the probability of failure measured by random testing was 0.975 for the 25 injected faults. As seen in Table 2, trials 9 and 16 caused a large disparity between ($\hat{\varepsilon}_{lG'D} \cdot \hat{\psi}_{alG'D}$) and $\hat{\tau}_{G'D}$. This is because perturbation functions create simulated infections that are independent of specific faults; $\hat{\tau}_{G'D}$ is caused by a specific fault, $\hat{\psi}_{alG'D}$ is not. Since perturbation functions attempt to create internal data state alterations that represent the internal state errors of many faults, it will sometimes occur that the propagation behavior observed from a specific fault will be different than the propagation behavior resulting from the use of a random number generator. Even though this is a drawback of this approximation technique, preliminary trials have suggested that propagation estimates are still frequently accurate. Additional data from other experiments is currently being collected.

6 Concluding Remarks

This paper has presented a technique based on the three necessary and sufficient conditions for software to fail. The *PIE* analysis technique dynamically estimates program characteristics that affect a program's computation. This technique does not need a specification nor oracle and may be performed on incorrect programs. We believe that the information collected about the effect of individual locations on the program's computational behavior has diverse applications: including where to emphasize testing resources, the degree of testing resources needed for this emphasis, and predicting software testability.

Plans are underway to automate *PIE* analysis. As mentioned in §3.5, there are obstacles into developing such a system, namely creating the sets of data states needed by infection and propagation analysis, and determining semantic equivalence between mutants and the original code. Regardless, a partially automated prototype for C programs that requires human intervention in certain circumstances is currently under development. This system will reduce tremendously the manual effort necessary to complete *PIE* analysis enormously, allowing *PIE* analysis to be applied to larger software systems.

Acknowledgements

The author thanks Larry Morell for his efforts during the development of these ideas. Appreciation is given to Larry Morell and Keith Miller for suggestions made on earlier drafts of this paper. Appreciation is also given to the anonymous referees for their insightful and invaluable comments.

trial	*location l*	*fault*
1	`TE := TBatts[Beta].[G].F[J]+E[J];`	omission fault/missing assignment
2	`TBatts[Beta][G].X := Batts[Beta][G].X+m* Batts[Beta][G].V*w* cos(Army[Beta][G].Theta);`	omission fault/missing assignment
3	`TBatts[Beta][G].NumSend:= NumSend+trunc(NumSend*j*AF);`	omission fault/missing assignment
4	`AF := 1/(Army[Beta][G].Squadrons-ND);`	omission fault/missing assignment
5	`m2 := Atan ((m2-m1)/(1+m1*m2));`	omission fault/missing assignment
6	`Term := Term*Army[not Beta][E].CommJamEff* (Army[not Beta][E].CommJamRadius - Dist(Batts[not Beta][E].X, Batts[not Beta][E].Y, Batts[Beta][G].X, Batts[Beta][G].Y)) / Army[not Beta][E].CommJamRadius;`	omission fault/missing assignment
7	`C := C - ixy;`	omission fault/missing assignment
8	`Tbatts[Beta]G.BK := Tbatts[Beta]G.BK +1;`	omission fault/missing assignment
9	`ND := ND + 1;`	omission fault/missing assignment
10	`Temp := Temp + Term;`	omission fault/missing assignment
11	`XCur := XCur + Army[Beta][G].Squadsep;`	omission fault/missing assignment
12	`Temp := Army[Beta][G].mweffect* Abs(Temp-wmaxseverity*numwevents)/ (wmaxseverity*numwevents;`	omission fault/missing assignment
13	`YCur := YCur -Army[Beta][G].Rowsep;`	omission fault/missing assignment
14	`Result := Result + Term;`	omission fault/missing assignment
15	`Utot[J] := Utot[J] + TBatts[Beta][G]. nw[e,j];`	omission fault/missing assignment
16	`Result := Abs(Result-params.numwevents* params.wmaxseverity)/ (params.wmaxseverity* params.numwevents))* Army[Beta][G].vweffect;`	omission fault/missing assignment
17	`XCorner := Batts[Beta][G].X - (((Army[Beta][G].Grow-1)/2)* Army[Beta][G].Squadsep);`	`XCorner := Batts[Beta][G].X - (((Army[Beta][G].Grow+1)/2)* Army[Beta][G].Squadsep);`
18	`A4 := (m+1)*((n+1)*Terrain[m,n]-n* Terrain[m,n+1])-m*((n+1)* Terrain[m+1,n]-n*Terrain[m+1, n+1]);`	`A4 := (m)*((n+1)*Terrain[m,n]-n* Terrain[m,n+1])-m*((n+1)* Terrain[m+1,n]-n*Terrain[m+1, n+1]);`
19	`Dist := sqrt(abs(sqr(x1-x)+ sqr(y1-y)));`	`Dist := sqrt(abs(sqr(x1-x)* sqr(y1-y)));`
20	`C1X := TX-TW/2;`	`C1X := TW-TX/2;`
21	`x1 := x+Batts[Beta][G].v* cos(Army[Beta][G].Theta);`	`x1 := x+Batts[Beta][G].v* sin(Army[Beta][G].Theta);`
22	`i := trunc(r);`	`i := trunc(round(r));`
23	`Xcorner := Batts[Beta][G].X- (((Maxi-1)/2)*Army[Beta][G]. Squadsep);`	`Xcorner := Batts[Beta][G].X- (((Maxi-1)*2)*Army[Beta][G]. Squadsep);`
24	`TempI := 1+(k+m) mod mod Batts[Beta][G].NW[E,J];`	`TempI := (k+m) mod mod Batts[Beta][G].NW[E,J];`
25	`Af := Army[Beta][G].FixRate* Batts[Beta][G].Numfixers/cc;`	`Af := Army[Beta][G].FixRate+ Batts[Beta][G].Numfixers/cc;`

Table 1: Injected Faults

trial	*a* variable	$\hat{\varepsilon}_{lG'D}$ execution estimate	$\hat{\psi}_{alG'D}$ propagation estimate	$\hat{\tau}_{G'D}$ failure probability estimate	$\hat{\varepsilon}_{lG'D} \cdot \hat{\psi}_{alG'D}$
1	`TE`	1.0	1.0	1.0	1.0
2	`TBatts[Beta][G].X`	1.0	1.0	0.97	1.0
3	`TBatts[Beta][G].NumSend`	1.0	0.67	1.0	0.67
4	`AF`	1.0	1.0	1.0	1.0
5	`m2`	0.98	0.49	0.58	0.48
6	`Term`	0.98	1.0	0.98	0.98
7	`C`	0.97	0.134	0.06	0.13
8	`Tbatts[Beta]G.BK`	1.0	0.99	1.0	0.99
9	`ND`	0.59	0.07	0.08	0.041
10	`Temp`	1.0	0.95	1.0	0.95
11	`XCur`	1.0	1.0	1.0	1.0
12	`Temp`	1.0	1.0	1.0	1.0
13	`YCur`	1.0	1.0	1.0	1.0
14	`Result`	0.27	0.0	0.0	0.0
15	`Utot[J]`	0.03	0.0	0.0	0.0
16	`Result`	0.95	0.421	0.14	0.39
17	`XCorner`	1.0	1.0	1.0	1.0
18	`A4`	0.98	1.0	0.85	0.98
19	`Dist`	1.0	0.98	1.0	0.98
20	`C1X`	0.98	0.0	0.08	0.0
21	`x1`	0.98	1.0	0.98	0.98
22	`i`	1.0	1.0	0.98	1.0
23	`Xcorner`	0.16	0.8125	0.09	0.13
24	`TempI`	0.01	1.0	0.01	0.01
25	`Af`	1.0	0.87	0.94	0.87

Table 2: Probability Estimates

References

[1] Ian F. Blake. *An Introduction to Applied Probability.* John Wiley and Sons Inc., 1979.

[2] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.

[3] Larry J. Morell and Richard G. Hamlet. Error Propagation and Elimination in Computer Programs. Technical Report TR-1065, University of Maryland, Department of Computer Science, July 1981.

[4] Richard G. Hamlet. Testing Programs with Finite Sets of Data. *Computer Journal*, 20(3):232–237, August 1977.

[5] Richard G. Hamlet. Testing Programs With the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[6] Richard G. Hamlet. Probable Correctness Theory. *Information Processing Letters*, pages 17–25, April 1987.

[7] Willam E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[8] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis.* McGraw-Hill Book Company, 1982.

[9] Bodgan Korel. Dynamic Program Slicing. *Information Processing Letters*, October 1988.

[10] Bodgan Korel. PELAS-Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, SE-14(9), September 1988.

[11] Bodgan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, SE-16(9):870–879, August 1990.

[12] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[13] Stephen K. Park and Keith W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[14] J. Voas and L. J. Morell. Applying Sensitivity Analysis Estimates to a Minimum Failure Probability for Software Testing. In *Proc. of the 8th Pacific Northwest Software Quality Conf.*, pages 362–371, Portland, OR, October 1990. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.

[15] L. J. Morell. A Model for Code-Based Testing Schemes. *Fifth Annual Pacific Northwest Software Quality Conf.*, pages 309–326, 1987.

[16] L. J. Morell. Theoretical Insights into Fault-Based Testing. *Second Workshop on Software Testing, Validation, and Analysis*, pages 45–62, July 1988.

[17] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, SE-16, August 1990.

[18] Larry Joe Morell. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[19] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

[20] A. J. Offutt. The Coupling Effect: Fact or Fiction. *Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification*, December 1989. Key West, FL.

[21] S. K. Park. Lecture notes on simulation, version 3.0. Department of Computer Science, College of William and Mary in Virginia, 1990.

[22] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[23] Timothy J. Shimeall. `CONFLICT` Specification. Technical Report NPSCS-91-001, Computer Science Department, Naval Postgraduate School, Monterey, CA, October 1990.

[24] D. Richardson and M. Thomas. The RELAY Model of Error Detection and its Application. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.

[25] J. Voas. *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs.* PhD thesis, College of William and Mary in Virginia, March 1990.

[26] Martin D. Davis and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science.* Academic Press, 1983.

[27] Steven J. Zeil. Testing for Perturbations of Program Statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.